Rewriting rule-based model for aspect-oriented software evolution

Hanene Cherait* and Nora Bounour

LISCO Laboratory,
BadjiMokhtar – Annaba University,
P.O. Box 12, 23000 Annaba, Algeria
Email: hanene_cherait@yahoo.fr
Email: nora_bounour@yahoo.fr
*Corresponding author

Abstract: Software is evolutionary in nature. From the time a software product is defined until it is no longer used, it changes. We focus in this paper on the aspect-oriented (AO) software evolution. Although AO software engineering is the subject of ongoing research, AO software evolution has received less attention. AO programming is a mature technology that modularises the crosscutting concerns. Unfortunately, it produces new dependencies between them; restricts the evolvability of the software system. In order to cope with all types of AO program's dependencies, we converge toward a new evolution modelling approach. In our proposal, the AO source code is modelled in a more abstract and formal format as an attributed coloured graph, where the different dependencies in the software system are well defined. Then, the change requests are presented as rewriting rules on this coloured graph. We give here, the details of our approach as well as its implementation. And, we provide an empirical evaluation to prove the efficiency of our proposal.

Keywords: aspect-oriented programming; AOP; software evolution; reverse-engineering; graph rewriting, source code modelling

Reference to this paper should be made as follows: Cherait, H. and Bounour, N. (xxxx) 'Rewriting rule-based model for aspect-oriented software evolution', *Int. J. Computer Applications in Technology*, Vol. X, No. Y, pp.xxx–xxx.

Biographical notes: Hanene Cherait is a PhD student in Complex Software Engineering. She received her Master of Science degree in Computer Science from the University of Badji Mokhtar – Annaba (UBMA), Algeria in 2009. Her research interests include software evolution; aspect-oriented programming and software reverse engineering.

Nora Bounour received her Doctorate degree in the Department of Computer Science at the University of Badji Mokhtar – Annaba (UBMA), Algeria in the year 2007. She is presently working in the same department as an Associate Professor. She is the Head of the research group on reengineering and evolution of complex systems at the Laboratory of Complex System Engineering (LISCO). Her research interests include software evolution and reverse engineering methodologies, separation of concerns and aspect-oriented programming.

1 Introduction

Software evolution refers generally to progressive change in the software's properties or characteristics (Lehman and Belady, 1985). Managing software change is the key process in software evolution. This process of change in one or more of their attributes leads to the emergence of new properties or to improvement, in some sense (Lehman and Ramil, 2007). Therefore, software evolution should not be treated by an ad hoc or unstructured process (Lehman and Belady, 1985). This is why, a lot of techniques and more specifically models have been proposed to manage the software evolution in a reliable and organised way along its life cycle. The research efforts spent in the software evolution topic show the importance of the domain (Pan et al., 2013; Hammad et al., 2014).

Although there are a large number of evolution techniques for the different programming paradigms (procedural software, object-oriented software, …, etc.), seldom effort has been spent for aspect-oriented (AO) software evolution. To overcome this problem is a hot topic. Aspect-oriented programming (AOP) (Kiczales et al., 1997) is a technique for modularising crosscutting concerns. AspectJ (The AspectJ Team, 2012) is the most popular AO language. It constitutes an extension of Java. The AspectJ program can be divided into two parts: base code which includes classes, interfaces and other language constructs as in Java, and aspect code which includes aspects for modelling crosscutting concerns in the program. Each aspect which is woven at a specific set of join points is solely responsible for a particular crosscutting concern (Kiczales et al., 1997).

In spite of the more advanced modularisation mechanisms (Suganthi and Nadarajan, 2013), AO programs still suffer from evolution problems. According to the study of Tourwé et al. (2003), "current aspect oriented software design (AOSD) technologies deliver applications that are as hard, or perhaps even harder, to evolve than was the case before…". One observed characteristic of AOP is that it results in a large number of additional (coarse-grained to fine-grained) system units (aspects) ready to be composed to the final application. With this growing number of system units, the dependencies between them become vast and tangling (crosscutting relationships are difficult to depict cleanly and effectively). Aspects are not explicitly invoked but instead, are implicitly invoked (Xu et al., 2004). So, changes introduced with AOP are not visible directly in the base system's source code, making program comprehension more difficult. Aspects are usually stored in separate files; but the effects of this code can influence the whole system (Vollmann, 2002). To resume up, modelling AO program evolution involves more complex relationships than in the traditional ones (object-oriented, procedural…).

The aim of our paper is to propose and illustrate an efficient evolution model for AO software, and specifically, for the AspectJ source code. The main contributions of this paper are summarised as follows:

- Modelling the AO source code: In our proposal, the AO (AspectJ) source code is modelled as a coloured graph (Heckel et al., 2002). Our model represents an AO system at an abstraction level of the system components and the different relations between them.
- Change modelling: Define rules to model changes to the AO system. Describing what kind of changes can be made on the model. These changes are formalised as rewriting rules (Ehrig et al., 2006) on the coloured graph.
- A prototype tool that automates the reverse-engineering of the AspectJ source code to a coloured graph.

The rest of the paper is organised as follows. In the next section, we give the background used in this paper. Section 3 presents the overview of our approach. Section 4 gives the model (coloured graph) representing the AO source code. In Section 5, we explain how change requests can be formulated as rewriting rules. Section 6 details our tool validation as well as our experimentations. We pass briefly on the related work in Section 7. Finally, we conclude our discussion and present the future work in Section 8.

2 Background

Before we detail our evolution model for AO software, some basic background has to be presented. This section provides the basic concepts of an AO source code and more specifically AspectJ programming language, and then we pass briefly on the foundations of the algebraic graph rewriting formalism.

2.1 Foundations of the AspectJ language

In this paper, we use AspectJ as our target language to show the basic idea of our evolution model for AO software. The choice of AspectJ is motivated by its wide popularity, mature language design, industrial-strength tool support and by its nature as an extension of Java. Before presenting our approach, we first briefly introduce the background of AspectJ semantics. More information about AspectJ can be found in The AspectJ Team (2012).

The decomposition of an AspectJ application makes appear:

- The base code that defines the set of the services (i.e., functionalities) achieved by the application. In other words, the code corresponds to the 'what' of the application.
- Several complementary aspects that specify the mechanisms governing the execution of the application, i.e., the non-functional aspects defining the 'how' (synchronisation, persistence or security). The aspects are used to regroup choices of implementation that have an impact on the whole system and that would be scattered otherwise through the whole code.

To illustrate the fundamental concepts of AspectJ language, we refer to the example in Figure 1. This program changes the monitor to refresh the display as needed. It contains an aspect *UpdateDisplay* that updates the display when objects move, and a class *Point*.

Figure 1 The program UpdateDisplay (see online version for colours)

```
Class Point {
  private int _x = 0,
              _y = 0;
  int getX(){
    return _x;
  }
  int getY(){
    return _y;
  }
  void setX(int x){
     _x = x;
  }
  void setY(int y){
    _y = y;
  }
}
```

```
aspectUpdateDisplay{
  public String Point.name ;
  public void Point.setName
  (String name){
    this.name = name;
  }
  public String
  Point.getName(){
    return name;
  }
  pointcut move():
    call(void
    Point.setX(int))||
    call(void Point.setY(int));
  before(): move() {
    System.out.println("figure
    is going to be displaced");
  }
  after(): move (){
    Display.update();
  }
}
```

Table 1 Kinds of join points

Join point	*Pointcut designator*	*Description*
Method call	call(*MethodPattern*)	When a method is called
Method execution	execution(*MethodPattern*)	When the method's body is executed
Constructor call	call(*ConstructorPattern*)	When a constructor is called
Constructor execution	execution(*ConstructorPattern*)	When a constructor' s body is executed
Static initialiser execution	staticinitialisation(*TypePattern*)	When the static initialisation of a class is executed
Object pre-initialisation	preinitialisation(*ConstructorPattern*)	Before the initialisation of the object
Object initialisation	initialisation(*ConstructorPattern*)	When the initialisation of an object is executed
Field reference	get(*FieldPattern*)	When a non-constant attribute of a class is referenced
Field set	set(*FieldPattern*)	When an attribute of a class is modified
Handler execution	handler(*TypePattern*)	When a treatment of an exception is executed
Advice execution	adviceexecution()	When the code of an advice is executed

AspectJ provides new features for modularisation, in particular when adding new functionality to an existing system.

The novel features can be classified into two groups, one influence the dynamic behaviour of the base code by injecting new code when certain events occur in its execution: join point, pointcut, and advice. The second group of features allows one to statically add new members to classes: introduction. So, AspectJ extended Java through additional keywords to support AOP concepts:

- *Join points*: they are the places where the crosscutting actions take place. They represent well-defined points in the execution of a program, such as method calls, object field accesses and so on. The purpose of the AspectJ language is allowing the programmer to precisely and succinctly identify and manipulate join-points. Table 1 describes the kinds of join points, for every join point a specific pointcut designator is used.
- *Pointcut*: after we identify the join points useful for a crosscutting functionality, we need to select them using the pointcut construct. A pointcut is a Boolean expression over a fixed set of predicates and the operators; and (&&), or (||) and not (!). It selects join points based on a given criteria, such as method names and so on (Table 1), and collects context at those points. Pointcuts serve to define which advice has to be applied.
- *Advice*: an advice represents a program module which is to be executed at the designated join points. There are three types of advices *before*, *after* (*after returning*/*after throwing*) and *around*, which correspond to the program modules to be executed prior, after or instead of the designated events, respectively. The body of an advice is much like a method body – it encapsulates the logic to be executed upon reaching a join point. In contrast to the methods of traditional object-oriented languages, advices are not called explicitly. Instead, the execution of an advice is automatically 'triggered' when the control flow reaches the join point that is designated (Vollmann, 2002).
- *Introduction (inter-type declaration)*: introductions are used to crosscut the static type structure of classes. That is, they insert additional class members like constructors, methods, and fields into classes as if they were declared in the classes themselves. They may even change the class's super-class and its super-interface, respectively.
- *Aspect*: it is the central unit in AspectJ, in the same way that a class is the central unit in Java. It contains the code that expresses the weaving rules for both dynamic and static crosscutting. Additionally, aspects can contain data, methods, and nested class members, just like a normal Java class. Every aspect is destined to be developed in an independent way then integrated to an application by a process called 'aspect weaving'.

If we compare the above concepts with the program in Figure 1, we distinguish: one aspect named 'UpdateDisplay'. This last contains a pointcut declared with the name 'move'. Which specifies two join points of the type *Method call*; 'when the method *setX* is called', and 'when the method setY is called' (class Point's methods).

The two are joined with the operator 'or (||)'. And two advices (*before*, *after*). Besides, it contains three introductions: it introduces an attribute 'name' and two methods 'setName', 'getName' to the class 'point'.

2.2 Algebraic graph rewriting

The main idea of the algebraic approach to graph rewriting (Ehrig et al., 2006) is to give an abstract algebraic characterisation of attributed coloured graphs (Heckel et al., 2002). A coloured graph can be formalised as follows:

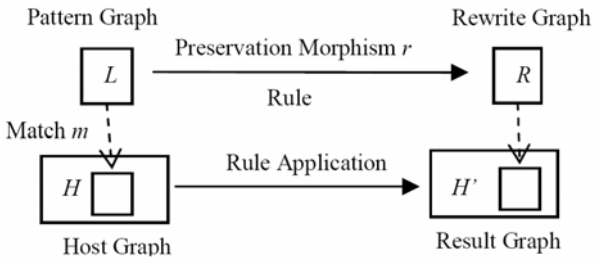
- *Coloured graph*: Formally, a coloured graph, for example, G, is represented by a 6-tuple as: $G = \{N_G; A_G; s_G; t_G; m_1; m_2\}$. Here, N_G denotes the set of nodes, AG denotes the set of edges; s_G is a mapping that maps the edges to their sources and t_G maps them to their targets. m_1 and m_2 are mappings that map the nodes and the edges in the graph to the fixed alphabets of node and edge colours, respectively. The colours are very important to give a semantic description of the

element (node, edge). Without names (colours) it would be hard to identify the elements and their interrelationships.

- *Graph rewriting*: The calculus of graph transformation (graph rewriting) has a solid background. In this paper we present only as much theoretical background as needed to understand our approach. From Figure 2 (Blomer et al., 2012), a graph rewrite rule consists of a tuple $L \rightarrow R$, whereas L the left hand side (LHS) of the rule is called pattern graph and R the right hand side (RHS) of the rule is the replacement graph. Moreover, we need to identify graph elements (nodes or edges) of L and R for preserving them during rewrite. This is done by a preservation morphism r mapping elements from L to R; the morphism r is injective, but needs to be neither subjective nor total (Blomer et al., 2012).

The transformation is done by the application of a rule to a host graph H. To do so, we have to find an occurrence of the pattern graph in the host graph. Mathematically speaking, such a match m is an isomorphism from L to a sub graph of H. This morphism may not be unique, i.e., there may be several matches.

Figure 2 Basic idea of graph rewriting



Afterwards we change the matched spot m (L) of the host graph, such that it becomes an isomorphic sub graph of the replacement graph R. Elements of L not mapped by r are deleted from m (L) during rewrite. Elements of R not in the image of r are inserted into H; all others (elements that are mapped by r) are retained. The outcome of these steps is the resulting graph H'. The rewrite rules can be specified with a textual (Blomer et al., 2012; Glauert et al., 1997) or graphical (Schultzke and Ermel, 2012) manner.

Potentially, additional constraints as to when a rule should or should not be applicable may be specified as attribute conditions (ACs), or negative application conditions (NACs). A NAC is a graph pattern which must not be present in the host graph for the rule to be deemed applicable. ACs are Boolean expressions which may appear in order to express constraints on attribute values. The labels prefixing the nodes and edges must be used for identification purposes, meaning a node (or edge) bearing the same label in LHS and RHS (and possibly NAC) refers to the same node (or edge) in the host graph (e.g., Figure 7).

A set of graph rewriting rules, together with a type graph (Corradini et al., 1996), is called a graph transformation system (GTS). One of the main static analysis facilities for GTSs is the check for conflicts and dependencies between rules and transformations. We argue that the existing theoretical results for graph transformation can advantageously be used for analysing potential conflicts and dependencies in AO software evolution.

3 Overview of our approach

In this section of the paper, we present the overview of our proposed evolution model for AspectJ source code. AO software evolution can be defined as the process of progressively modifying the elements of an AO software system in order to improve or maintain its quality over time. The main idea of our proposal is to accurately model how this software evolves by using a more abstract and formal format.

We use graph transformation (graph rewriting) (Ehrig et al., 2006) as a formal technique to give a formal semantics to our evolution model and to analyse it rigorously. The analogy between AO software evolution and graph transformation is quite natural: an AO software system can be expressed as a graph containing a set of components interrelated by connectors. Graph transformations allow us to express the evolution of these software graphs in a precise way. In addition, it enables formal analysis and reasoning about AO software evolution. Mehner et al. (2009) state that "The formal technique helps to make the problems explicit. It directs the developer to the problematic parts of a model. It helps in understanding aspect-oriented compositions and it helps in reasoning effectively about the crosscutting".

Figure 3 Overview of our approach (see online version for colours)

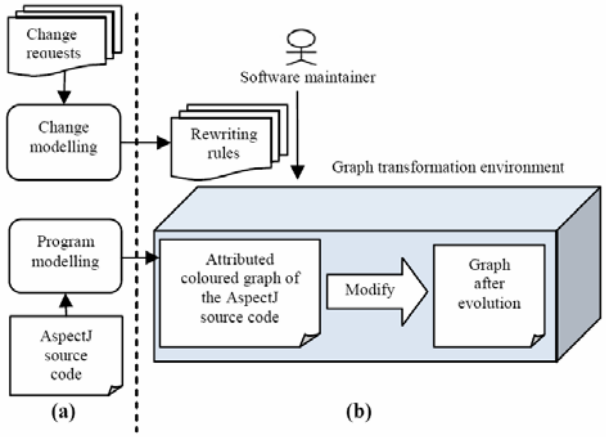


Figure 3 depicts the overview of our approach, which can be divided in two main steps:

1 as presented in Figure 3(a), the AspectJ source code is represented as an attributed coloured graph (Heckel et al., 2002), and the changes to the software are formalised as rewriting rules that transform the graph G to a graph G'; in order to achieve the evolution requests

2 the software maintainer modifies the coloured graph of the AspectJ source code by applying sequences of rewrite rules in a certain order [Figure 3(b)].

The changes are presented in a formal format as rewriting rules. In contrast to the text format of the change, the rewrite rule is more accurate and meaningful, because it contains the full information about the change:

1 Where it is applied: the pre-condition of the rule gives the parts of the graph that will be subject to change, i.e., the changed elements and their related entities.

2 How it is applied: the post-condition of the rule shows the altered parts of the graph after application of he rule.

3 When it is applied: the different conditions that we can formulate for the applicability of the rule (NAC, AC, …, etc.). If one of the conditions is not verified, the rule (change) will not be applied.

This rich format facilitates the comprehension of the change (where, how and when), and thereafter the software evolution.

The obvious first step towards our goal for representing AO software evolution by graph rewriting is the introduction of a suitable graph representation of AspectJ programs: one needs to decide which entities are represented by nodes, which relationships are represented by edges, and which information is represented by properties of nodes and edges. This is discussed in the next section.

4 Program modelling

At the first step our model aims to define an abstract representation of the AspectJ source code. In contrast to the text representation of the source code, the abstraction gives a more comprehensible and global overview of all software artefacts and their relationships eliminating the fine-grained details. This facilitates the evolution tasks: change impact analysis, change propagation, …, etc.

In our approach, the AspectJ source code is modelled by an attributed coloured graph (Heckel et al., 2002). This last is generated directly from the AspectJ source code. We feel that for our purpose, the graph-based approach is very suitable. In view of the wide acceptance of graph-like representations in modelling software, it seems natural and interesting to use graph rewriting as the basis for the desired evolution model.

Graphs are based on a well understood mathematical foundation ‘graph theory’ (Godsil and Royle, 2001). This makes them very interesting from a formal point of view. From a practical point of view, graphs are also very useful, since they are used often as an underlying representation of arbitrarily complex software artefacts and their interrelationships (Mens, 2001).

In order to formally specify the AspectJ source code as a coloured graph, we should have to present the meta-model of the proposed ‘coloured graph’. This meta-model guarantee the consistency of the model (graph) to every transformation, which specifies what it means for a model to be valid (well formed). We use therefore; a type graph (Corradini et al., 1996) that plays the role of a meta-model. This type graph (class diagram), shown in Figure 4, specifies how to create well-formed coloured graph of AspectJ software. Any well-formed AspectJ source code can be represented as a graph that conforms to this type graph.

We can detect that this type graph is the union of two parts: the base code sub-graph and the aspect(s) subgraph(s). These two parts are related with different dependence edges. In the following, we give the details of the different elements of the AspectJ type graph (meta-model).

4.1 The base code sub-graph

The base code of the AspectJ source code is an object oriented program. This last consists of source code entities and their dependencies. In our representation, the nodes represent the entities of the program and the edges are theirs dependencies. As discussed before (Section 2.2), an attributed coloured graph has a pair of colour alphabets, one to colour the edges and one to colour the nodes. The elements of the node colour alphabet for the base code are the following:

- *Class*: «Name», «Visibility» (is one of: abstract, private, protected or public).
- *Attribute*: « Name», «Visibility», « Type»
- *Method*: «Name», «Visibility»
- *Parameter*: «Name», « Type»
- *Return value*: « Type».

Furthermore, the edges describe the relationships (dependencies) between these entities as presented in Table 2. The relationships can be classified into three classes:

- The first class of relationships depicts which attributes and methods belong to a class. It is also important for this model to show the parameters and the return values that methods take or return.
- The second class of relationships shows the connection between classes. Here, *relationship* is one of *association*, *aggregation*, *generalisation*, or *composition*.
- The third class of relationships captures the object relations (calls). This type of relationship is important to include in our model because they specify which methods and parameters are affected by the changes caused by evolution requests.

Figure 4 Type graph of the AspectJ model (see online version for colours)

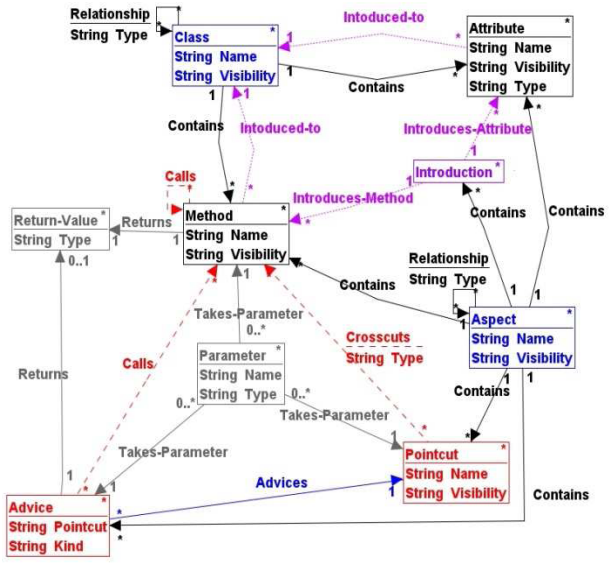


Table 2 Edge colours of the base code sub-graph

Source node	*Target node*	*Edge colour*	*Attribute*
Class	Attribute	Contains	-
	Method	Contains	-
	Class	Relationship	Type
Parameter	Method	Takes-parameter	-
Method	Return value	Returns	-
	Method	Calls	-

4.2 The aspect sub-graph

An aspect is an encapsulation unit. It consists also of entities and dependencies. Every aspect of the system is also modelled with a coloured sub-graph. This last is similar to the sub-graph that models the base code program, but we must add other concepts proper to the AspectJ source code. The elements of the node colour alphabet are:

- *Aspect*: «Name», «Visibility»
- *Attribute* : « Name», «Visibility», « Type»
- *Method*: «Name», «Visibility»
- *Parameter*: «Name», « Type»
- *Return value*: « Type »
- *Pointcut*: «Name», «Visibility»
- *Advice*: « Pointcut», « Kind» (is one of: before/after returning/after throwing/after/around)
- *Introduction*.

Table 3 groups the different dependencies in the aspect sub-graph. For every edge, we define its source/target node, as well as the colour used and the attribute(s) needed.

4.3 Modelling the global system

In order to model the entire AspectJ source code, we must give a global overview of the different entities of the program and theirs dependencies.

Table 3 Edge colours of the aspect sub-graph

Source node	*Target node*	*Edge colour*	*Attribute*
Aspect	Attribute	Contains	-
	Method	Contains	-
	Pointcut	Contains	-
	Advice	Contains	-
	Introduction	Contains	-
	Aspect	Relationship	Type
Method	Method	Calls	-
	Return value	Returns	-
Parameter	Method	Takes-parameter	-
	Pointcut	Takes-parameter	-
	Advice	Takes-parameter	-
	Pointcut	Advices	-
Advice	Advice around → Return value	Returns	-
	Method	Calls	-
Introduction	Attribute	Introduces-attribute	-
	Method	Introduces-method	-

To integrate the different sub-graphs that represent the classes and the aspects, we need three types of edges, Table 4 describes these edges:

- *Crosscuts*: this type of edges links the pointcut of the aspect with its join point(s). They must show the information about the join point type: method call/method execution… (Table 1).
- *Introduced-to*: this type of edges links the methods or attributes introduced by the aspect with the class(s) where they are introduced to.
- *Calls*: these edges present the calls between aspect's methods/advices to the base code methods, i.e., a method (advice) of an aspect can use the methods of a class to perform a specific treatment. Besides of the constraints between nodes and edges presented above, the type graph (Figure 4) expresses the following constraints on the instance graphs:

  1 *Constraints of multiplicity on the edges*: For example, every pointcut or advice is contained in precisely only one aspect. A class contains zero or several attributes and methods, the method has one return value and the return value is related to one method, …, etc.

2 *Constraints of multiplicity on the nodes*: In the type graph considered, we did not define constraints of multiplicity on the nodes. However, we could have decided to permit the graphs that contain at least a node of type aspect (while attaching the cardinality of 1..* to the aspect), and so on.

3 *Constraints of attributes*: The specificities of the components of the program, as the types are represented mainly on the edges and the nodes by attributes that are mainly string of characters.

An AspectJ source code can now be specified formally as a graph conforming to the type graph, together with all of its graph constraints. An example is given in Figure 5. It shows a graph representing the AspectJ program 'UpdateDisplay' of Figure 1 (this graph has been generated by our convertor tools, see Section 6). In this figure, we can see that there are three sub-graphs: in the LHS of the figure, we depict the coloured graph of the class *point*, and in the RHS the coloured graph of the aspect UpdateDisplay. The two sub-graphs are related with five dependence edges: two of the type crosscut and three of the type introduced-to. At the bottom of the figure, we depict the sub-graph of the class 'display', its method update is called by the advice after move (call edge).

Table 4 Dependence edges

Edge colour	*Source node*	*Target node*	*Attribute*
Calls	Aspect's method	Class's method	-
	Advice	Class's method	-
Crosscuts	Pointcut	Class's method	Type
Introduced-to	Method, attribute introduced by the aspect	Class	-

4.4 Discussion

The representation of the AspectJ source code as a unique graph does not avoid the modularity of the code. It is very important to preserve the separation of crosscutting concerns when modelling the AO program, the primary motivation behind AOP. Our proposal maintains strict separation of base-code and crosscutting concerns in the proposed source-code model, i.e., every concern (aspect) is modelled as a coloured graph. The weaving is presented as dependence arcs between the base code sub-graphs and aspect sub-graphs. So, it is very easy to distinguish between the base code and the crosscutting concerns (aspects) of the software. In summary, the proposed evolution model for AspectJ source code preserves the modularity of the aspect-oriented paradigm, at the same time it makes more visible and clear the dependencies between the crosscutting concerns of the system, which is note an easy task with the plain text representation of the AspectJ source code.

5 Change modelling

The aim of our work is not just to formalise the evolution operations but to *automate* their application too. While AspectJ source code is graphically formalised by coloured graph, evolution requests (changes) are mapped to graph rewriting rules. A rewrite rule changes graphically (automatically) the sets of entities and dependencies of a program to evolve it; where the entities are considered as nodes and the dependencies are the edges between the program entities. Like graphs, graph rewriting is very intuitive in use. Nevertheless, it has a firm theoretical basis. These theoretical foundations of graph rewriting can assist in proving correctness and convergence properties of the AO software evolution. We represent changes to the program as explicit rewriting rules to its coloured graph. When a change rewrite rule is applied it takes as input a program state and returns an altered program state.

We have two types of change operations:

1 *Atomic change operations*: The basic evolution operations (changes) that will serve as foundations to the creation of more complex operations. The main atomic change operations are the addition and the deletion operations, since we believe that change operation can be modelled by deleting the old entity (or dependency) and adding the new one.

   Consequently, the atomic rewriting rules are:

   - *addition rule*: add a new node (edge) to the program graph
   - *deletion rule*: deletes an existing node and all its dependencies, or the deletion of just an edge.

2 Composite change operations: the atomic change operations can be combined to realise various evolution requests. The combination of several basic operations will be able to give birth to other evolution operations, or to an evolution process. For example, an evolution request that requires moving a pointcut from aspect A to aspect B consists in deleting it from A and adding it to B. Besides, we have to move the advice(s) related with this pointcut to aspect B also, or we just delete it (them) according to the change request. These atomic changes can be grouped in a single 'move pointcut change'. This is practically performed via the *rule sequence* concept. This last is the union of all the rules formulated for a specific evolution request, applied in a certain order just like an algorithm.

   Starting from the definitions in the Section 2.2, we can give a precise and clear meaning for the graph rewriting system concepts that describe our AspectJ evolution model.

Table 5 gives the mapping of graph transformation concepts to the AspectJ software evolution.

Table 5 AspectJ software evolution as a GTS

Graph transformation concepts	*AspectJ evolution concepts*
Host graph	AspectJ coloured graph.
Rewriting rule	The evolution operation that presents the change, which must be done on the software.
Left hand side (LHS) of the rule	The sub-graph related to the change request, i.e., the entity (s) related to the change and their relationships.
Right hand side (RHS) of the rule	The sub-graph that presents the LHS after evolution. Modify the LHS in order to meet the change request.
LHS∩RHS	The graph part that must be unchanged, which is not touched by the evolution request.
LHS\ (LHS∩RHS)	The graph part which shall be deleted. Represent the elements touched by the evolution request.
RHS\ (LHS∩RHS)	The graph part which shall be created. The changed elements after evolution.
Sequence rule	The union of all the rules formulated for a specific evolution request, applied in a certain order.
Graph transformation system $GTS = (G_0, R)$	An evolution process of an AspectJ source code, where: G_0 is the starting graph; and R is a set of evolution operations (graph rewriting rules).

In the following, we present an evolution scenario for the graph in Figure 5 using the attributed graph grammar (AGG) tool (Schultzke and Ermel, 2012). We will change the program *UpdateDisplay* (Figure 1) to meet the following evolution requests:

a Do not display a message before the call of the methods *setX* and *setY* ⇒ delete the advice *before* () *move*.

b Control the value of x 'if x > 10 x = x-1' ⇒ add a new pointcut 'control' to capture the method *setX*:

```
pointcut control():
call (void Point.setX(int));
int around (int x): control(){
  if (x>10) x=x-1;
  return x;
}
```

c Change the name of the aspect UpdateDisplay to DisplayAndControl ⇒ modify the attribute Name of the aspect.

The rewriting rules for these requests are the following:

a *Deletion rule*: Figure 6 depicts the deletion rule. It deletes the advice executed *before* the pointcut *move*. The deletion of an entity involves the deletion of all their related dependencies (the edges between the advice *before* and the pointcut *move* and the aspect *UpdateDisplay* are deleted too).

Figure 5 The coloured graph of the program UpdateDisplay (see online version for colours)

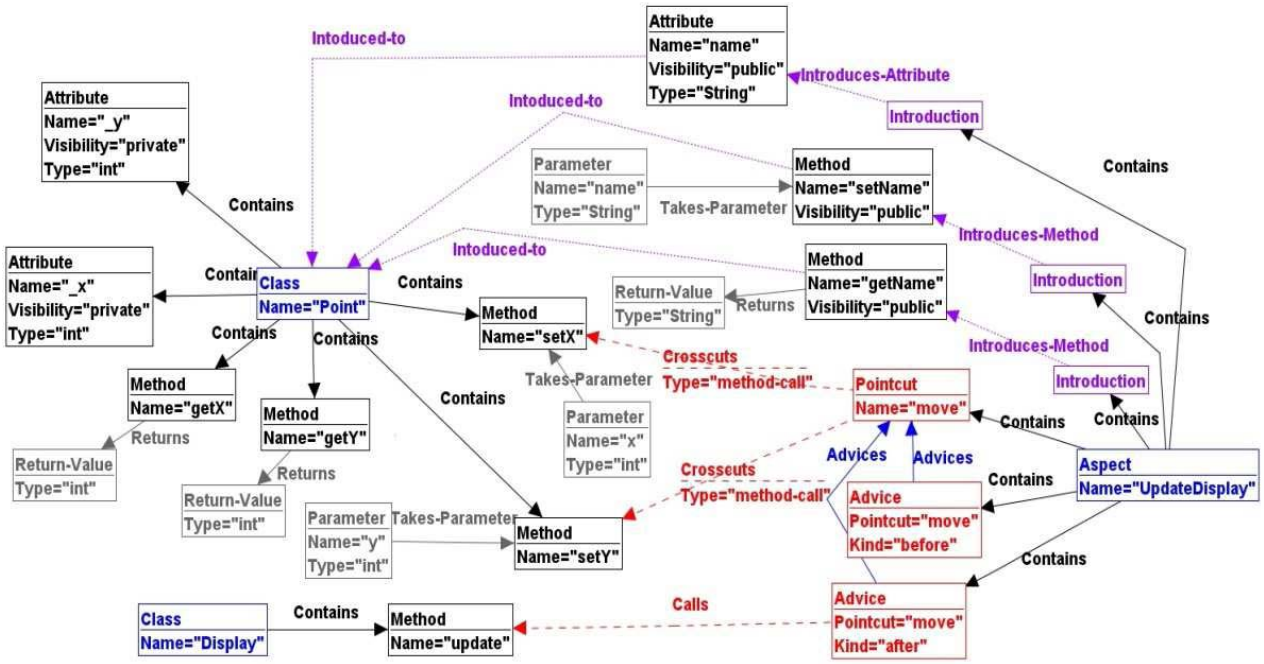


Figure 6 Delete the advice before move (see online version for colours)

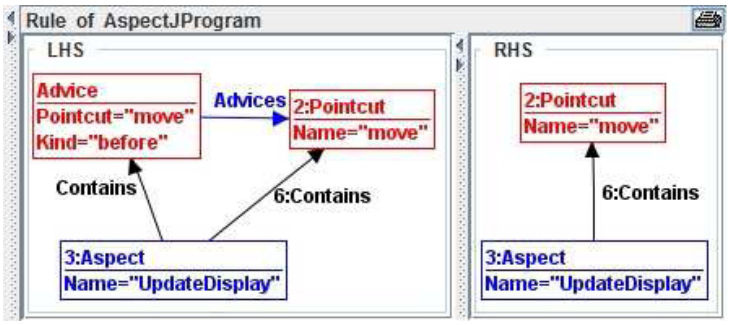


Figure 7 Add the pointcut control (see online version for colours)

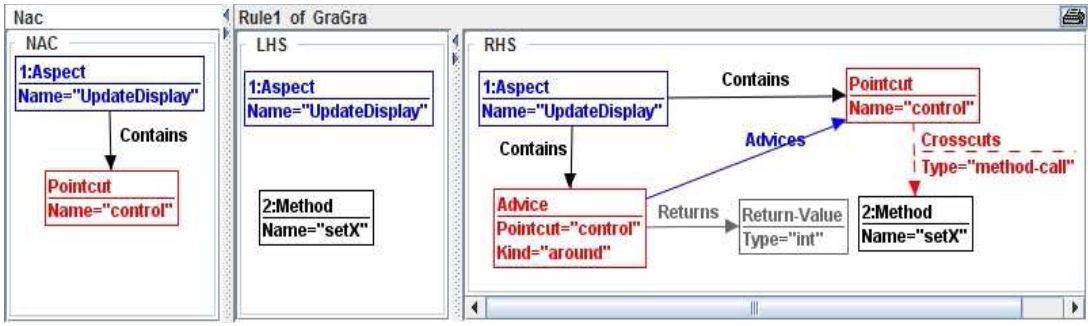
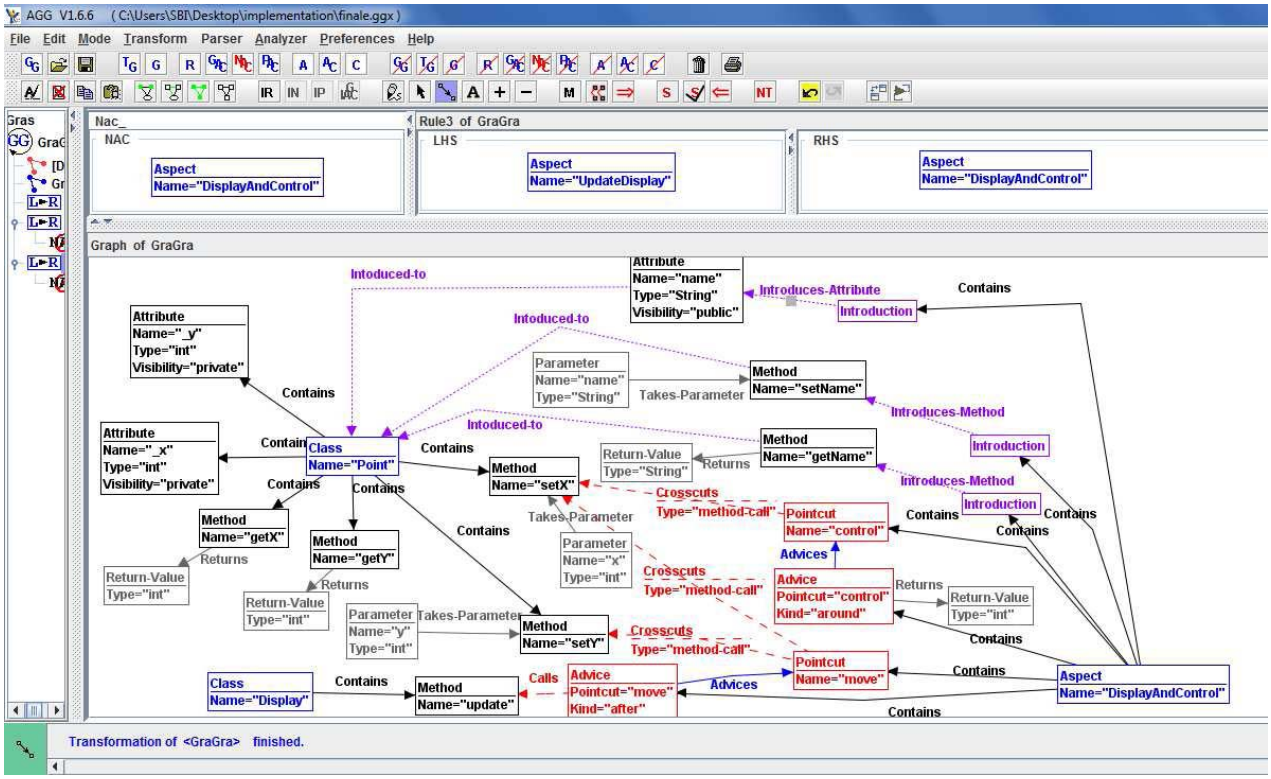


Figure 8 Graph after evolution (see online version for colours)



b *Addition rule*: The rule in Figure 7 adds the pointcut *control* detailed above, to the aspect *UpdateDisplay* 'if it is not already existed'. This condition is formulated using the NACs depicted in the left side of the figure (i.e., the existence of two advices with the same kind for the same pointcut is prohibited). The RHS of the rule depicts that the addition of the pointcut involves the addition of its advice (advice around and its return

value). The join points are specified with the crosscut edges. The attribute type of the crosscut edge specifies that the join point is of the type *method call*.

c *Modification rule*: The modification rule is depicted in the top of Figure 8. This rule substitutes the old attribute Name of the aspect by the new one. The NAC is used here to avoid the existence of other aspect with the same name. We can note here that the modification request is formulated as the deletion of the old element (node or edge) and the addition of the new one.

After the application of these rules, the screenshot in Figure 8 shows the graph represents the program *UpdateDisplay* after transformation (evolution).

6 Tool validation

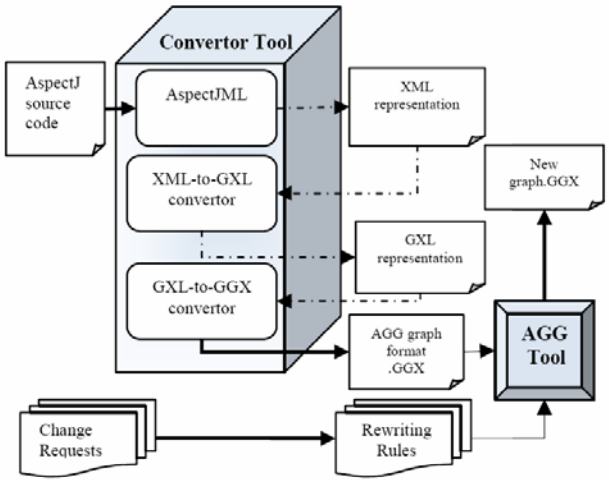
6.1 Tool overview

Our approach aims to reverse-engineer the AspectJ source code to a more abstract representation as an attributed coloured graph. The change requests are described as rewriting rules. These last are applied to the AspectJ graph via a graph transformation tool. So, the main problem in the validation of our proposal is the conversion of the AspectJ source code to a coloured graph representation. We have full automated this conversion. Figure 9 presents the overview of our tool validation, which can be resumed in the following parts:

- *Convertor tool*: to represent the AspectJ source code as an attributed coloured graph, we have implemented a convertor tool. This last can be divided in three main subconvertors.
  1 *AspectJML*: This is an existing open source proposed by Melo Junior and Mendonça (2005). It is an extended markup language (XML)-based markup language for representing source code written in AspectJ. The AspectJ source code is converted in *XML* format (Suzuki and Yamamoto, 1998) through the power of AspectJML.
  2 *XML-to-GXL convertor*: We have implemented this convertor. It converts the XML document produced by AspectJML to a *graph exchange language* (GXL) (Winter et al., 2001) modelling the AspectJ source code as a coloured graph. This convertor is an *XML stylesheet language transformation* (XSLT) document (Clark, 1999); where every AspectJ element is treated via a specific XSLT template. The XML-based representation (GXL) provides the interchangeability of AspectJ model information between various development tools such as CASE tools. This seamless tool interoperability increases our productivity to evolve an AspectJ source code.
  3 *GXL-to-GGX convertor*: We have implemented this convertor to convert the GXL document produced by the XML-to-GXL convertor to a graph grammar exchange (GGX). This last is the XML-based format used in the AGG (Schultzke and Ermel, 2012) tool to represent the host graph. This format is a GXL graph extended with layouts for nodes and edges. This convertor is an XSLT document too. It is based on the graph transformation environment used for the transformations. So, if we want to use another environment else than AGG, we just need to modify the convertor to generate the appropriate graph format of this environment (starting from the GXL graph).
- *AGG tool*: The change requests must be formalised as rewriting rules. Then, we use a graph transformation environment to apply these rules on the attributed coloured graph. In our validation, we used the AGG tool (Schultzke and Ermel, 2012), which is a powerful tool of graph transformation. We can formulate properties, constraints; analyse the graph, …, etc. The transformation of the AspectJ graph produces a new version of this one, where all the change requests are applied as rewriting rules.

Note: After the modification of the graph (evolution), we can regenerate the AspectJ source code following the inverse path: convert the GGX graph to GXL (via XSLT). This last must be converted in XML via XSLT, then in AspectJ via AspectJML. The use of XSLT is very interesting for our proposal. The processing of an XSLT stylesheet is very speed; we can generate the GXL (or GGX) document of a large application in an average of less than 10 seconds. So, our validation strategy is very efficient for smaller as well as large AspectJ source code.

Figure 9 Tool validation (see online version for colours)



6.2 Experimentation

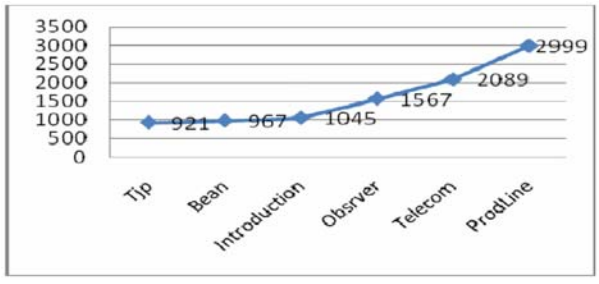
In order to assess the feasibility and correctness of *our approach*, some AspectJ programs were analysed and represented by the proposed graph-representation using our prototype tool. Our study used six AspectJ programs as shown in Table 6. Five of them are taken from AspectJ example package. The rationale behind was, this collection of programs has also been used as benchmarks by a lot of researches as case studies. Our experimentation also includes a benchmark used in many research works 'ProdLine'.

These benchmarks give us a variety of situations to validate our prototype tool. For example, the Benchmark Telecom contains ten classes and 39 methods; ProdLine contains 11 aspects and 84 introductions, 15 pointcuts, and so on. The applicability of our approach to these case studies shows its feasibility to represent and validate AO software evolution. We verified the coloured graphs (GXL documents) generated by our tool against a manual inspection of the graph and the associated analysed source code for each of aforementioned programs.

Our experiments showed that the coloured graphs generated by the tool were correct for smaller benchmarks (e.g., Tjp) as well as the large ones (e.g., ProdLine).

The execution time needed for the generation of graphs is dependent with the size of the AspectJ source code as well as the number of their modules. The graph in Figure 10 shows the execution time (in millisecond) for the generation of GXL graphs for every previous Benchmark. So, our representation of AO software provides a useful support for gaining a better knowledge of the internal structure of these complicated programs, by reducing the effort needed for obtaining them in a variety of software engineering tasks, and especially in AO software evolution.

Figure 10 Execution time in millisecond (see online version for colours)



So, we can use our prototype as a reverse-engineering tool of AspectJ source code.

7 Related work

This section of the paper presents related works discussing the benefits of our proposal in contrast to the other ones. Our work involves the following research areas:

- *Graph-based modelling for AO software*: A variety of graph-based models have been proposed to represent the different features of the AO programs (Bernardi and Di Lucca, 2007; Lemos et al., 2007; Parizi and Abdul Ghani, 2008; Zhao, 2003). Each of these models puts the accent on some of the specific features of a program for slicing or testing purposes, *not for evolution propose*. Most of them are fine-grained representations, which represent a program at the 'statements' level, i.e., nodes represent statements and arcs represent the different dependencies between them. However, the tendency in the evolution techniques is to use coarse-grained representations, that put in evidence the different components of the artefact (e.g., aspects, advices, methods) and the dependencies between them (e.g., membership, crosscutting). Our approach follows this principle to model the AO software evolution.
- *Aspect-oriented software evolution*: It is still an emerging research area; this is largely due to the fact that few large-scale AO software systems exist today. This is why evolution models for AO software systems do not exist yet, hence, our proposal. However, many techniques (Chavez et al., 2009; Griswold et al., 2006; Kellens et al., 2006; Pires et al., 2011) exist to treat the fragile pointcut problem. For instance, Kellens et al (2006) propose a model-based pointcuts. They decouple the pointcut definitions from the actual structure of the base program, and define them in terms of a conceptual model of the software instead. We believe that our abstract representation of the AspectJ source code offered by our approach can be used complementary with these techniques to alleviate this problem.

Table 6 Analysed programs

Program	*#Classes*	*#Aspects*	*#Method*
Telecom	10	3	39
Bean	2	1	16
Observer	6	2	9
Tjp	1	1	5
Introduction	1	3	13
ProdLine	9	11	10
Program	*#Pointcut*	*#Advice*	*#Introduction*
Telecom	6	6	3
Bean	1	2	1
Observer	1	1	11
Tjp	2	1	-
Introduction	-	-	6
ProdLine	15	15	84

Our model makes more visible and clear the dependencies between the crosscutting concerns of the system as well as all the dependencies between the aspects and the base code, i.e., if we change any join point, we can detect the crosscutting concerns (aspects) or more specifically the

pointcut(s) related to this join point. Using one of the previous techniques, we can verify that the pointcuts are well implemented in our model, and consequently, in the AO source code.

8 Conclusions

We proposed in this paper an evolution model for AO source code written in AspectJ. It is based on the algebraic graph rewriting formalism which gives it a formal background and an automatic implementation method (employing graph rewrite tools). Our approach starts by reverse engineering the source code to a coloured graph; representing the different entities of the system and their dependencies. The evolution requests are formalised using rewriting rules on the system graph. We can combine several rules to achieve different evolution requests of an AO software system. A prototype tool is built and case studies are experimented to demonstrate the feasibility of our approach. Although this is not the scope of this paper, we believe that this approach is general enough to be applicable to other AO programming languages, i.e., even if AO languages may require specific kinds of nodes and edges, they can be all expressed using the same notation as an attributed coloured graph.

References

Bernardi, M.L. and Di Lucca, G.A. (2007) 'An interprocedural aspect control flow graph to support the maintenance of aspect oriented systems', in *ICSM'07: Proceedings of IEEE International Conference on Software Maintenance*, pp.435–444.

Blomer, J., Geib, R. and Jakumeit, E. (2012) *The GrGen.NET User Manual* [online] http://www.grgen.net (accessed 3 February 2012).

Chavez, C., Garcia, A., Batista, T., Oliveira, M., Sant'Anna, C. and Rashid, A. (2009) 'Composing architectural aspects based on style semantics', in *AOSD'09: Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development*, pp.111–122.

Clark, J. (1999) *XSL Transformations (XSLT) Version 1.0*, Recommandation 16, November [online] http://www.w3.org/TR/xslt (accessed 31 January 2012).

Corradini, A., Montanari, U. and Rossi, F. (1996) 'Graph processes', *Fundamenta Informaticae*, Vol. 26, Nos. 3 and 4, pp.241–265.

Ehrig, H., Ehrig, K., Prange, U. and Taentzer, G. (2006) 'Fundamentals of algebraic graph transformation', *EATCS Monographs in Theoretical Computer Science*, Springer, ISBN: 978-3-540-31187-4.

Glauert, J.R.W., Kennaway, R., Papadopoulos, A.G. and Sleep, R. (1997) 'Dactl: an experimental graph rewriting language', *Journal of Programming Languages*, Vol. 5, No. 1, pp.85–108.

Godsil, C. and Royle, G.F. (2001) 'Algebraic graph theory', *Graduate Texts in Mathematics*, Springer, ISBN: 978-0-387-95220-8.

Griswold, W.G. et al. (2006) 'Modular software design with crosscutting interfaces', *IEEE Journal of Software*, Vol. 23, No. 1, pp.51–60.

Hammad, M., Hammad, M. and Bsoul, M. (2014) 'An approach to automatically enforce object-oriented constraints', *Int. J. of Computer Applications in Technology*, Vol. 49, No. 1, pp.50–59.

Heckel, R., Kuster, J.M. and Taentzer, G. (2002) 'Confluence of typed attributed graph transformation systems', in *Proceedings of ICGT'02, LNCS*, Springer-Verlag, Barcelona, Spain, Vol. 2505, pp.161–176.

Kellens, A., Mens, K., Brichau, J. and Gybels, K. (2006) 'Managing the evolution of aspect-oriented software with model-based pointcuts', in *ECOOP'06: Proceedings of the 20th European Conference on Object-Oriented Programming, LNCS*, Springer-Verlag, Nantes, France, Vol. 4067, pp.501–525.

Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M. and Irwin, J. (1997) 'Aspect-oriented programming', in *ECOOP'97: Proceedings of 11th European Conference on Object-Oriented Programming, LNCS*, Springer-Verlag, Vol. 1241, pp.220–242.

Lehman, M.M. and Belady, L. (1985) *Program Evolution – Processes of Software Change*, Academic Press Professional, Inc., San Diego, CA, USA, 538 pp., ISBN: 0-12-442440-6.

Lehman, M.M. and Ramil, J.F. (2007) 'Software evolution and software evolution processes', *Annals of Software Engineering*, Vol. 14, Nos. 1–4, pp.275–309.

Lemos, O.A.L., Vincenzi, A.M.R., Maldonado, J.C. and Masiero, P.C. (2007) 'Control and data flow structural testing criteria for aspect-oriented programs', *Journal of Systems and Software*, Vol. 80, No. 6, pp.862–882, Elsevier.

Mehner, K., Monga, M. and Taentzer, G. (2009) 'Analysis of aspect-oriented model weaving', *Transactions on AOSD 5, LNCS*, Springer-Verlag, Vol. 5490, pp.235–263.

Melo Junior, L.S. and Mendonça, N.C. (2005) 'AspectJML: a markup language for AspectJ', in *WASP'05: Proceedings of the 2nd Brazilian Workshop on Aspect Oriented Software Development*, Uberlândia, MG, Brazil.

Mens, T. (2001) 'A Formal foundation for object oriented software evolution', in *ICSM'01: Proceedings of 17th IEEE International Conference on Software Maintenance*, Florence, Italy, pp.549–552.

Pan, W., Jiang, B. and Xu, Y. (2013) 'Refactoring packages of object-oriented software using genetic algorithm based community detection technique', *Int. J. of Computer Applications in Technology*, Vol. 48, No. 3, pp.185–194.

Parizi, R.M. and Abdul Ghani, A.A. (2008) 'AJcFgraph-AspectJ control flow graph builder for aspect-oriented software', *International Journal of Electrical and Computer Engineering*, Vol. 3, No. 3, pp.170–181.

Pires, P.F., Delicato, F.C., Pinto, M., Fuentes, L. and Marinho, É. (2011) 'Software evolution in AOSD: a MDA-based approach', in *Proceedings of CBSE'11*, Boulder, Colorado, USA, pp.193–197.

Schultzke, T. and Ermel, C. (2012) *AGG Environnement: A Short Manual*, Short Manual edition, User Manual [online] http://tfs.cs.tuberlin.de/agg/ShortManual.ps (accessed 15 March 2012).

Suganthi, S. and Nadarajan, R. (2013) 'Role of aspect-oriented approach in dynamic adaptability', *Int. J. of Computer Applications in Technology*, Vol. 47, No. 4, pp.334–342.

Suzuki, J. and Yamamoto, Y. (1998) 'Managing the software design documents with XML', in *Proceedings of the 16th Annual International Conference on Computer Documentation*, ACM Press, New York, pp.127–136.

The AspectJ Team (2012) *The AspectJ Programming Guide*, Online manual [online] http://eclipse.org/aspectj/ (accessed 31 January 2012).

Tourwé, T., Brichau, J. and Gybels, K. (2003) 'On the existence of the AOSD evolution paradox', in *AOSD'03: Proceedings of Workshop on Software-engineering Properties of Languages for Aspect Technologies*, Boston, USA.

Vollmann, D. (2002) 'Visibility of join-points in AOP and implementation languages', in *AOSD'02: Proceedings of Second Workshop on Aspect-Oriented Software Development*, Bonn, Germany, pp.65–69.

Winter, A., Kullbach, B. and Riediger, V. (2001) 'An overview of the GXL graph exchange language', in *Proceedings of International Seminar Dagstuhl Castle, LNCS*, Springer-Verlag, Germany, Vol. 2269, pp.324–336.

Xu, J., Rajan, H. and Sullivan, K. (2004) 'Understanding aspects via implicit invocation', in *ASE'04: Proceedings of 19th IEEE International Conference on Automated Software Engineering*, pp.332–335.

Zhao, J. (2003) 'Data-flow-based unit testing of aspect-oriented programs', in *COMPSAC'03: Proceedings of 27th Annual IEEE International Computer Software and Applications Conference*, Dallas, Texas, pp.188–197.